# ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

### **SCUOLA DI INGEGNERIA E ARCHITETTURA**

### Dipartimento di Informatica – Scienza e Ingegneria

### Corso di Laurea Magistrale in Ingegneria Informatica

### **Progetto finale ed attività progettuale**

### in

### SISTEMI DIGITALI M

# Briscola con Altera DE1 Board



CANDIDATI:

|  |  |  |
| --- | --- | --- |
| Lorenzo Mustich | Alessandro Paoletti | Alessandro Morabito |
| 0000901917 | 0000901759 | 0000907048 |

Anno Accademico: 2019/2020

# Sommario (da inserire solo alla fine)

[Capitolo 1. La Briscola 2](#_Toc18877272)

[Capitolo 2. Architettura del sistema 2](#_Toc18877273)

[Capitolo 3. Datapath 2](#_Toc18877274)

[Capitolo 4. Control Unit 2](#_Toc18877275)

[Capitolo 5. Graphic User Interface: il lato Java 2](#_Toc18877276)

[Capitolo 6. Punto d’incontro: comunicazione seriale 2](#_Toc18877277)

[Capitolo 7. Audio Errore. Il segnalibro non è definito.](#_Toc18877278)

[7.1 Codec stereo WM8731 **Errore. Il segnalibro non è definito.**](#_Toc18877279)

[7.2 Protocollo I2C **Errore. Il segnalibro non è definito.**](#_Toc18877280)

[7.3 Interfaccia PCM **Errore. Il segnalibro non è definito.**](#_Toc18877281)

[7.4 Entità di top-level **Errore. Il segnalibro non è definito.**](#_Toc18877282)

[Capitolo 8. Fonti 2](#_Toc18877283)

# Capitolo 1. La Briscola

*“La briscola. Gioco molto semplice. L’avversario sbatte sul tavolo una carta, e voi dovete sbatterla più forte. I buoni giocatori rompono dai quindici ai venti tavoli a partita. […] Quando la carta è abbastanza vecchia, diventa molto dura e pesante, e se non siete allenati è opportuno giocare con guanti da elettricista”*.

Così nel 1976, lo scrittore bolognese Stefano Benni definiva la briscola nel suo “*Bar Sport*” (Mondadori, 1976). Gioco di carte praticato a tutte le latitudini della Penisola e dalle origini incerte, consiste nel totalizzare un numero di punti più alto rispetto agli altri giocatori.

Si gioca con un mazzo di 40 carte con semi italiani. Alla prima mano, il mazziere di turno distribuisce tre carte ciascuno e lascia una carta scoperta sul tavolo trasversalmente al mazzo, la quale segnerà il seme della briscola e sarà l’ultima ad essere pescata. I punti assegnati ad ogni carta sono così ripartiti:

* Asso -> 11 punti,
* 3 -> 10,

definiti *carichi*;

* Re -> 4,
* Cavallo -> 3,
* Fante (o Donna) -> 2,

dette *figure*;

* 7, 6, 5, 4, 2 –> 0,

detti *lisci*.

Ad inizio partita, il giocatore alla destra del mazziere tira per primo; durante le mani successive, questa prerogativa sarà data al giocatore vincitore della mano precedente.

Il primo giocatore del giro detta, con la propria carta lanciata, il seme della mano: gli altri giocatori potranno “prendere” (lanciare per primi alla successiva mano ed, eventualmente, ottenere dei punti se le carte prese non sono dei lisci) solo se la propria carta messa sul tavolo ha lo stesso seme, punteggio maggiore o, in caso di parità, numero maggiore. Ad ogni turno, i giocatori devono avere sempre tre carte in mano. Il punteggio massimo ottenibile è di 120.

Il numero di giocatori è variabile: dall’uno contro uno alla possibilità di giocare in coppie da due o da tre eliminando via via un numero di carte lisce maggiori (di solito si scartando i due) per evitare che avanzino delle carte.

BriscolaDE1 permette una sfida secca e diretta tra un giocatore reale ed uno virtuale.

# Capitolo 2. Algoritmi di gioco

Dall’analisi del problema è scaturita la possibilità di suddividere il gioco in due fasi:

* *fase 1*:
* *fase 2*:

Il penultimo turno richiede un’analisi approfondita a parte in quanto, a seconda del valore dell’ultima briscola, i giocatori potrebbero attuare strategie diverse rispetto a quelle usate nel corso degli altri turni:

# Capitolo 3. Architettura del sistema

## 3.1 Entità di top-level

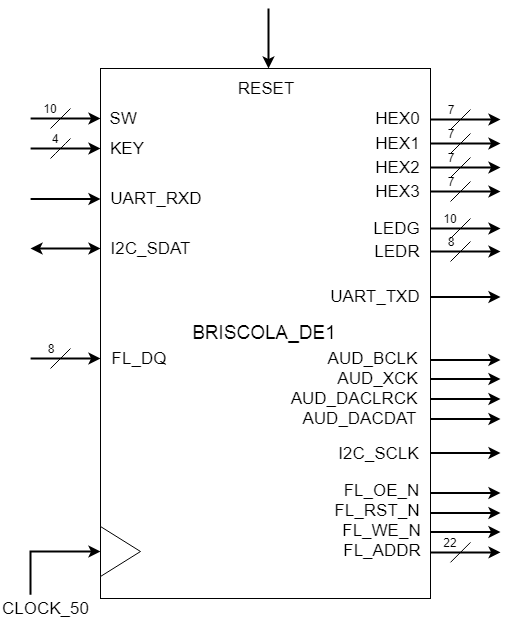
L’*entità di top-level* rappresenta l’entità più astratta di tutto il sistema, la quale si interfaccerà direttamente con la scheda DE1. È stata definita nel file *Briscola.vhd*.

Figura 1. Entità di top-level

Di seguito sono illustrati i pin fisici e il loro utilizzo:

* *CLOCK\_50:* uno dei tre clock generati dall’FPGA, utilizzato per sincronizzare l’intero sistema;
* *RESET:* reset fornito dalla DE1;
* pin per l’interfacciamento con l’utente e per il debug:
  + *HEX0*, *HEX1*, *HEX2*, *HEX3:* quattro display a sette segmenti utili in fase di debug per la visualizzazione del valore di segnali, parametri numerici delle carte e identificatore numerico dello stato della Control Unit;
  + *SW:* dieci switch, di cui attivi solo SW(9) per il RESET e SW(7) per l’attivazione e lo spegnimento dell’audio;
  + *LEDG*, *LEDR:* dieci led verdi e otto rossi, su cui visualizzare, in fase di debug, i bit ricevuti ed inviati all’applicazione Java;
  + *KEY:* quattro tasti utilizzati per abilitare la trasmissione di dati sulla linea seriale;
* pin per la comunicazione seriale:
  + *UART\_RXD* per la ricezione;
  + *UART\_TXD* per la trasmissione;
* pin per la riproduzione dell’audio:
  + *AUD\_BCLK:* clock per la trasmissione dei dati audio;
  + *AUD\_XCK:* clock per la sincronizzazione del codec audio;
  + *AUD\_DACLRCLK:* il clock per la sincronizzazione del DAC;
  + *AUD\_DACDAT:* dati utilizzati dal codec;
* pin per l’utilizzo del protocollo I2C utile alla configurazione del codec audio:
  + *I2C\_SDAT:* dati I2C;
  + *I2C\_SCLK:* clock I2C;
* pin per la lettura da FLASH:
  + *FL\_DQ:* dati letti;
  + *FL\_OE\_N:* ouput enable (logica negativa);
  + *FL\_WE\_N:* write enable (logica negativa);
  + *FL\_RST\_N:* reset asincrono (logica negativa);
  + *FL\_ADDR:* indirizzo della cella di memoria letta;

## 3.2 Architettura

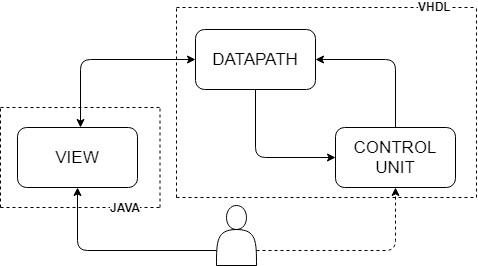
Il *sistema* è basato sulla seguente architettura:

Figura 2. Architettura del sistema

L’*utente* interagisce con un’interfaccia implementata in Java, la quale, attraverso uno scambio di informazioni tramite protocollo seriale, comunica ai moduli VHDL le intenzioni del giocatore. In alcuni punti della partita sarà possibile (a volte necessario) relazionarsi direttamente con l’FPGA.

I moduli principali sono:

* *Briscola\_Datapath*: implementa le unità di calcolo e i registri utili al funzionamento del sistema (capitolo );
* *Briscola\_Controller*: si configura come un automa a stati finiti; ha il compito di coordinare tutte le azioni del datapath tramite l’invio e la ricezione di segnali appositi (capitolo );

ai quali si aggiungono:

* *PLL*: circuito *phase-locked loop* integrato alla scheda; a partire da un clock da 50 MHz, genera due clock da 12 (timing codec audio, capitolo ) e da 50 MHz collegati ad ogni altro modulo del sistema;
* *UART\_RX*: implementa l’interfaccia UART necessaria per la comunicazione seriale;
* *Briscola\_Audio*: permette l’utilizzo del codec audio WM8731;

## 3.3 Packages

Nel progetto sono stati inseriti vari package in cui sono stati definiti i tipi di dato, le costanti e le funzioni statiche utilizzate poi in tutti gli altri moduli:

* *briscola\_package*: definisce i tipi di dato su cui si basa tutto il progetto;
* *briscola\_audio\_package*: contiene le costanti utili al funzionamento delle entità relative alla parte audio;
* *briscola\_datapath\_package*: raccoglie le dichiarazioni e le definizioni delle function legate al datapath;
* *briscola\_utility\_package*: contiene le dichiarazioni e le funzioni di utilità;

Durante l’analisi del problema, è fuoriuscita la possibilità di suddividere il gioco in varie *fasi* o *“situazioni”*; da qui, la necessità di definire un package per ognuno di essi contenenti dichiarazione ed implementazione di varie funzioni legate al giocatore virtuale:

* *briscola\_lisci\_package*: da utilizzare se nella mano sono presenti dei lisci;
* *briscola\_situazioneNoLisci\_package*: da utilizzare se nella mano non vi sono dei lisci;
* *briscola\_fase2\_package*: da utilizzare quando il giocatore virtuale deve controbattere ad una mossa del giocatore fisico;
* *briscola\_penultimo\_turno\_package*: da utilizzare quando si è arrivati alla penultima mano della partita.

### **3.3.1 briscola\_package**

-- Costanti --

-- numero di turni totali di una partita (40 carte, 2 giocatori, 20 turni)

constant NUM\_TURNI: integer := 20;

-- Tipi di dato --

type vincitore is (GIOCATORE, CPU);

type seme is (BASTONI, DENARI, COPPE, SPADE);

type carta is record

numero : integer;

seme\_carta : seme;

valore : integer;

briscola : boolean;

end record;

type mazzo is array (0 to 39) of carta;

type mano\_cpu is array (0 to 2) of carta;

### **3.3.2 briscola\_audio\_package**

-- AUDIO CONSTANTS –

constant LAST\_FLASH\_ADDR : positive := 1661635;

constant AUDIO\_PRESCALER\_MAX : positive := 250;

constant I2C\_PRESCALER : positive := 60;

### **3.3.3 briscola\_datapath\_package**

-- audio costant

function decidiCarta(mano : mano\_cpu) return integer;

-- audio costant

function valutaPresa(carta1 : carta; carta2 : carta) return boolean;

### **3.3.4 briscola\_utility\_package**

-- dato un numero, genera il suo codice equivalente per un display a 7 segmenti

function numberTo7SegmentDisplay(numero : integer) return std\_logic\_vector;

-- dato un carattere, genera il suo codice equivalente per il display a 7 segmenti

function digitTo7SegmentDisplay(carattere : character) return std\_logic\_vector;

-- dato il numero della carta, restituisce il suo valore

function getValorefromNumber(numero : integer) return integer;

-- inverte l'ordine dei bit in un qualsiasi vettore

function reverse\_vector(a : std\_logic\_vector) return std\_logic\_vector;

-- controlla se un vettore è pieno di zeri

function vectorIsNotZero(vector : std\_logic\_vector) return boolean;

-- data la carta, la trasforma nel corrispondente byte

function fromCartaToByte(cart : carta) return std\_logic\_vector;

### **3.3.5 briscola\_lisci\_package**

-- determina se nella mano sono presenti dei lisci

function isLiscio(mano : mano\_cpu) return boolean;

-- restituisce vero se è presente un carico nella mano, falso altrimenti

function isCarico(mano : mano\_cpu) return boolean;

-- restituisce vero se esistono un liscio e un carico con lo stesso seme, falso altrimenti

function isCartaStessoSemeCarico(mano : mano\_cpu) return boolean;

-- restituisce l'indice della carta liscia più bassa

function getCartaLisciaPiuBassa(mano : mano\_cpu) return integer;

-- restituisce l'indice della carta liscia dello stesso seme nel carico

function getCartaStessoSemeCarico(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP-LEVEL --

-- restituisce l'indice della carta liscia più adeguata

function getCartaLiscia(mano : mano\_cpu) return integer;

### **3.3.6 briscola\_situazioneNoLisci\_package**

-- SITUAZIONE 1 --

-- restituisce l'indice della carta più bassa tra quelle in mano

function getCartaPiuBassa(mano : mano\_cpu) return integer;

-- restituisce l'indice della carta più bassa dello stesso seme di un carico

-- altrimenti restituisce la più bassa in assoluto

function situazione1(mano : mano\_cpu) return integer;

-- SITUAZIONE 2 --

-- restituisce l'indice della carta che non è un carico (no liscio)

function situazione2(mano : mano\_cpu) return integer;

-- SITUAZIONE 3 --

-- restituisce vero se nella mano ci sono più carichi dello stesso seme, falso -- altrimenti

function isCarichiStessoSeme(mano : mano\_cpu) return boolean;

-- restituisce l'indice del carico dello stesso seme più basso

function getCaricoStessoSemePiuBasso(mano : mano\_cpu) return integer;

-- restituisce vero se nella mano sono presenti degli assi, falso altrimenti

function getIndiceAssi(mano : mano\_cpu) return integer;

-- in caso di più carichi dello stesse seme, restituisce quello più basso

-- altrimenti, se ho assi restituisco l'indice del primo, altrimenti l'indice -- di una carta casuale

function situazione3(mano : mano\_cpu) return integer;

-- SITUAZIONE 4 --

-- restituisce l'indice della briscola

function getBriscola(mano : mano\_cpu) return integer;

-- restituisce vero se la briscola presente è con punti (sia figure che

-- carichi), falso altrimenti

function isBriscolaConPunti(mano : mano\_cpu) return boolean;

-- restituisce l'indice della carta non briscola e con pochi punti (una figura)

function getCartaNonBriscolaNonCarico(mano : mano\_cpu) return integer;

-- numero carichi: 1, numero briscole: 1

function situazione4(mano : mano\_cpu) return integer;

-- SITUAZIONE 5 --

-- restituisce l'indice della briscola più bassa

function situazione5(mano : mano\_cpu) return integer;

-- SITUAZIONE 6 –

-- restituisce l'indice del carico più alto

function getCaricoPiuAlto(mano : mano\_cpu) return integer;

-- numero carichi: 2, numero briscole: 1

function situazione6(mano : mano\_cpu) return integer;

-- SITUAZIONE 7 --

-- in presenza di tre briscole, restituisce l'indice della più bassa

function situazione7(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP\_LEVEL –

-- restituisce il numero di carichi in mano

function getNumeroCarichi(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP\_LEVEL --

-- restituisce il numero di briscole in mano

function getNumeroBriscole(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP\_LEVEL --

-- determina le varie situazioni in cui la CPU si può ritrovare in assenza di

-- lisci nella propria mano

function determinaSituazioneNoLisci(num\_carichi : integer; num\_briscole : integer; mano : mano\_cpu) return integer;

### **3.3.7 briscola\_fase2\_package**

--

function decidiCartaFase2(mano : mano\_cpu; cartaTerra : carta) return integer;

--

function getCartaNoBriscolaPresa(mano : mano\_cpu; cartaTerra : carta) return integer;

### **3.3.8 briscola\_penultimo\_turno\_package**

--

function getBriscolaPiuBassa(mano : mano\_cpu) return integer;

--

function getCartaValorePiuAlto(mano : mano\_cpu) return integer;

--

function getPrimaCartaLiscia(mano : mano\_cpu) return integer;

-- FUNZIONI DI TOP LEVEL

--

function decidiPenultimo(mano : mano\_cpu; briscola\_partita : carta) return integer;

--

function decidiPenultimoFase2(mano : mano\_cpu; briscola\_partita : carta; carta\_a\_terra : carta) return integer;

# Capitolo 4. Datapath

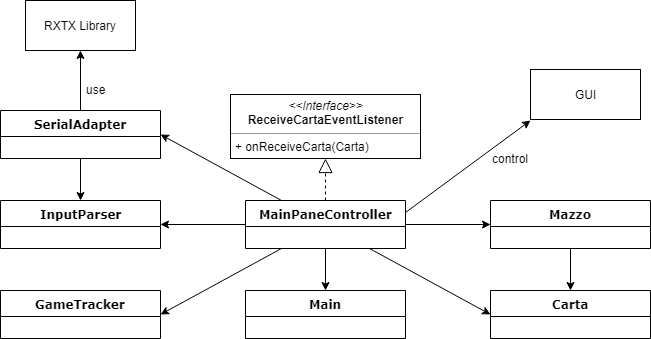
# Capitolo 5. Control Unit

# Capitolo 6. Graphic User Interface: il lato Java

La GUI si compone di un programma Java che utilizza classi relative al progetto in sé e componenti esterni.

## 6.1 Classi e relazioni tra le classi

* SerialAdapter: configurazione ed interfacciamento con la porta seriale. Gestisce l’invio dei dati sulla porta COM e scatena un evento ad ogni pacchetto ricevuto. I dettagli della configurazione sono gli stessi del lato VHDL  
  Metodi principali: **private** **void** readSerial()

  
Figura 3: Diagramma delle classi

**public** **void** writeToSerialPort(**byte** out)

* InputParser: trasformazione dei pacchetti in informazioni utili per il programma e viceversa; genera i byte delle carte e dei token da inviare nei pacchetti attraverso il SerialAdapter  
  Metodi principali: **public** **void** parseFrame(**byte** frame)

**public** **byte** fromCartaToByte (Carta c)

* GameTracker: traccia lo stato del gioco, contiene informazioni per la GUI, come chi detiene il turno, chi ha effettuato la presa della mano e genera il token in base allo stato del gioco.  
  Metodi principali: **public** **byte** getToken()
* MainPaneController: motore della GUI, incorpora tutti i metodi e le unità grafiche presenti nell’interfaccia, gestisce gli eventi scaturiti dall’utente e coordina tutte le altre classi per il funzionamento generale del sistema  
  Metodi principali: **public** **void** onReceiveCarta(Carta c)

**protected** **void** onClickPlayCard(ActionEvent e)

* Main: classe di base del progetto che carica l’interfaccia grafica e ne delega il controllo al Controller
* Mazzo: gestione delle funzioni del mazzo di carte: creazione del mazzo, gestione delle mani dei giocatori, distribuzione delle carte  
  Metodi principali: **private** ArrayList<Carta> creaMazzo()

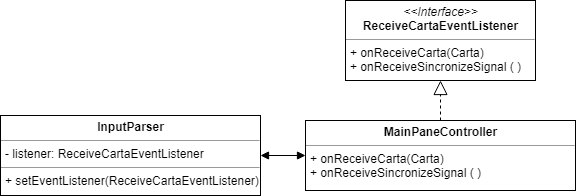
**public** Image getImageFromCarta(Carta c)

* Carta: contenuto informativo di ogni carta da gioco, compresa la sua immagine da visualizzare a schermo.

## 6.2 Pattern Observer

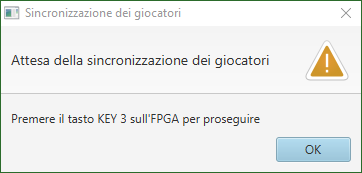
Il pattern Observer (Publish-Subscriber) viene utilizzato nei contesti in cui un oggetto richiede che altri oggetti aggiornino il proprio stato interno, notificandone il cambiamento.

In questa applicazione, il Controller della GUI crea la classe InputParser e successivamente chiama il metodo *setEventListener* passando come argomento sé stesso (***this***). Quando l’InputParser riceve una carta dalla porta seriale (evento notificato dal SerialAdapter), esso chiama il metodo *onReceiveCarta* dopo aver elaborato il frame ricevuto in un oggetto di tipo Carta.  
La presenza dell’interfaccia ReceiveCartaEventListener permette di rendere il codice riutilizzabile, cambiando solo l’implementazione dei metodi offerti dall’interfaccia.

  
Figura 4: Diagramma delle classi dell'applicazione del pattern Observer

## 6.3 Funzionamento

All’avvio della partita viene richiesto il nome del giocatore, che verrà poi visualizzato in fase di gioco.

  
Figura 5: Alert di attesa della sincronizzazione

In questa fase preliminare è necessario far comunicare direttamente i due componenti per poter instaurare una comunicazione corretta.

# Capitolo 7. Punto d’incontro: comunicazione seriale

L’utente interagisce con la View, scritta in Java, la quale notifica alla DE1 tutti i cambiamenti effettuati. La comunicazione avviene tramite protocollo seriale UART.

## 7.1 UART

# Capitolo 8. Audio

Per sfruttare al meglio le potenzialità della scheda, abbiamo deciso di inserire un sottofondo musicale che è possibile ascoltare durante la partita. La scelta è ricaduta su un frammento di liscio romagnolo *“Valzer romagnolo”* di Clementino (*IMG Edizioni*) per ricreare un’atmosfera conviviale tipica delle sagre e delle feste di paese.

## 8.1 Codec stereo WM8731

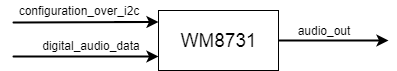
La DE1 contiene un codec stereo appositamente progettato per la lettura di file mp3, il WM8731.

Figura 6. Schema funzionamento codec audio

Come si evince dalla Figura 1, il codec legge dei dati audio digitali sotto forma di stringhe da 16 a 32 bit, campionandole con una frequenza variabile da un minimo di 8KHz ad un massimo di 96 KHz. Può funzionare sia come dispositivo MASTER che come SLAVE. Possiede un’interfaccia che fornisce:

* controlli di volume;
* possibilità di silenziare l’audio;
* possibilità di usare il codec in modalità stereo o mono

il tutto configurabile tramite il protocollo I2C.

La configurazione da noi usata per questo progetto è la seguente:

* codec in SLAVE mode;
* campionamento in USB mode:
  + clock a 12 MHz;
  + frequenza di campionamento di 48kSps;
  + risoluzione audio da 16 bit
* campionamento tramite PCM (*pulse-code modulation*);
* modalità mono

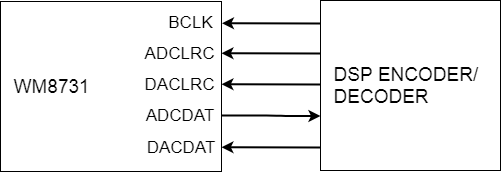
Il DSP (*Digital signal processor*), mostrato in figura 2, è un hardware dedicato e ottimizzato presente sulla scheda DE1 che permette di elaborare efficientemente segnali digitali.

Figura 7. WM8731 in SLAVE mode

Il WM8731, in SLAVE mode, riceve dal DSP il clock a cui tutto il sistema audio lavorerà (*BCLK, Digital Audio Bit Clock*) e i due clock per la sincronizzazione delle operazioni di conversione da analogico a digitale (*ADCLRC, ADC sample rate left/right clock*) e viceversa (*DACLRC, DAC sample rate left/right clock*); dopodichè, il codec stereo invia i dati audio da convertire (*pin* *ADCDAT, ADC digital audio data output*) e ne riceve la conversione (*pin* *DACDAT, DAC digital audio data output*).

## 8.2 Protocollo I2C

L’I2C (*Inter Integrated Circuit*), sviluppato nel 1982 dalla Philips, ma divenuto uno standard largamente diffuso solo nel 1992, è il protocollo che abbiamo scelto per la configurazione del codec audio.

Trattasi di un protocollo seriale che permette l’interfacciamento tra uno o più device *master* con un numero illimitato di altri device *slave*. Sono presenti solo due bus:

* SCL: *serial clock*, utilizzato per definire un clock unico per la connessione;
* SDA: *serial data*, per i dati effettivi;

entrambi i bus sono bidirezionali.

I device *master* generano il clock, danno inizio e fermano la connessione con i vari altri componenti e inviano i vari comandi permessi dal protocollo.

Una trasmissione dati base avviene in questo modo:

1. in principio, i due bus sono posti a livello alto;
2. il *master* genera una *Start condition* (SCL = 1, SDA = 0) seguito dall’indirizzo dello *slave* interessato: se il LSB è posto a 0, il master intende scrivere (*write mode*);
3. attende un *ack* da parte del device scelto;
4. in caso di *ack* positivo, invia un byte di dato;
5. attende un *ack* da parte del device scelto;
6. i punti 5 e 6 si ripetono fintanto che non venga generata una *Stop condition* (SCL = 1, SDA = 1).

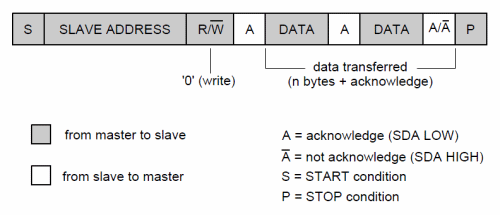
Di seguito è mostrato tramite l’uso di diagrammi temporali quanto appena spiegato.

Figura 4. Esempio di comunicazione I2C

Ai fini dell’implementazione di *BriscolaDE1*, è stata definita un’*entità* apposita che implementasse il protocollo scelto.

Figura 3. Funzionamento del protocollo I2C

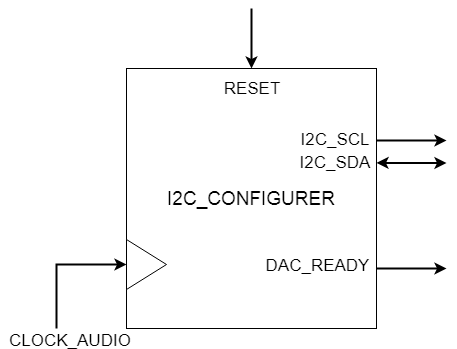
Il suo comportamento è stato definito utilizzando i seguenti *process*:

Figura 5. Entità I2C

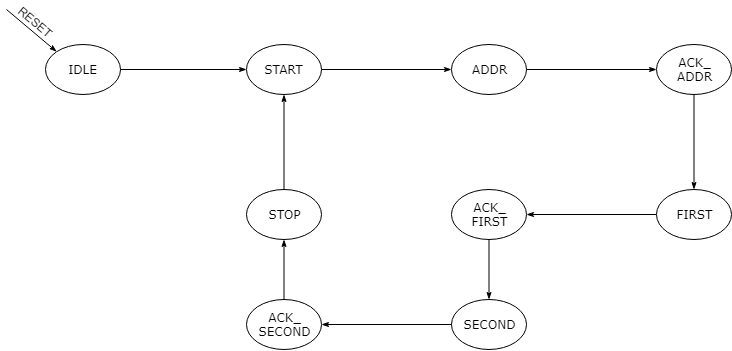
* I2CClockProcess : process(CLOCK\_12, RESET)
  + si occupa di generare, tramite l’utilizzo di un prescaler, i clock utili al protocollo e le transizioni tra i dati; DA METTERE CODICE
* FSMI2CProcess : process(CLOCK\_12, RESET)
  + definisce un automa a stati finiti che scandisce il regolare funzionamento del protocollo; a grandi linee, può essere visto nel seguente modo:

Figura 6. ASF dell’I2C\_codec

* InitProcess : process(CLOCK\_12, RESET)
  + si occupa della configurazione del codec WM8731

case init\_counter is

when 0 =>

-- reset

i2c\_data(15 downto 9) <= "0001111";

i2c\_data(8 downto 0) <= "000000000";

i2c\_send\_flag <= '1';

when 1 =>

-- active interface

i2c\_data(15 downto 9) <= "0001001";

i2c\_data(8 downto 0) <= "111111111";

i2c\_send\_flag <= '1';

when 2 =>

-- ADC off, DAC on, Linout ON, Power ON

i2c\_data(15 downto 9) <= "0000110";

i2c\_data(8 downto 0) <= "000000111";

i2c\_send\_flag <= '1';

when 3 =>

-- Digital Interface: DSP, 16 bit, slave mode

i2c\_data(15 downto 9) <= "0000111";

i2c\_data(8 downto 0) <= "000010011";

i2c\_send\_flag <= '1';

when 4 =>

-- headphone volume

i2c\_data(15 downto 9) <= "0000010";

i2c\_data(8 downto 0) <= "101111001";

i2c\_send\_flag <= '1';

when 5 =>

-- USB mode

i2c\_data(15 downto 9) <= "0001000";

i2c\_data(8 downto 0) <= "000000001";

i2c\_send\_flag <= '1';

when 6 =>

-- enable DAC to LINOUT

i2c\_data(15 downto 9) <= "0000100";

i2c\_data(8 downto 0) <= "000010010";

i2c\_send\_flag <= '1';

when 7 =>

-- remove mute DAC

i2c\_data(15 downto 9) <= "0000101";

i2c\_data(8 downto 0) <= "000000000";

i2c\_send\_flag <= '1';

init <= '1';

dac\_ready <= '1';

when others => … -- reset data and flag

end case;

## 8.4 Interfaccia PCM

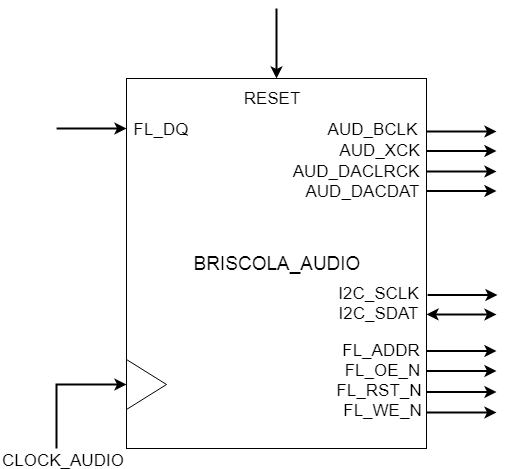
In base a quanto mostrato fino a questo punto, l’entità di top-level che si occuperà della configurazione del codec audio WM8731 (tramite l’utilizzo dell’I2C\_CONFIGURER) e della riproduzione dell’audio sarà la seguente:

Figura 7. Entità PCM

Per semplicità di progettazione, abbiamo deciso di suddividere i due compiti del componente in due *process* separati:

* AudioGenProcess : process(CLOCK\_AUDIO, RESET)
  + genera l’audio inviato al codec WM8731: si occupa di ricevere i dati audio e, una volta campionati, li invia al sistema audio della scheda DE1
  + DA METTERE CODICE
* ReadAudioProcess : process(CLOCK\_AUDIO, RESET)

legge il file audio dalla memoria FLASH DA METTERE CODICE

Per la memorizzazione del file, abbiamo optato per l’utilizzo delle memorie integrate alla scheda DE1, nel caso specifico abbiamo sfruttato la semplicità d’uso della memoria FLASH da 4Mb presente. Il file audio è stato salvato utilizzando il System Control Panel della DE1.

Il file audio “*liscio.wav*”, con risoluzione a 16 bit e campionamento a 48kHz, ha una dimensione di 1.58MB, per una lunghezza di 17 secondi. Ciò che sarà possibile ascoltare consiste in un loop del suddetto.

Figura 8. System Control Panel DE1

# Capitolo 9. Fonti